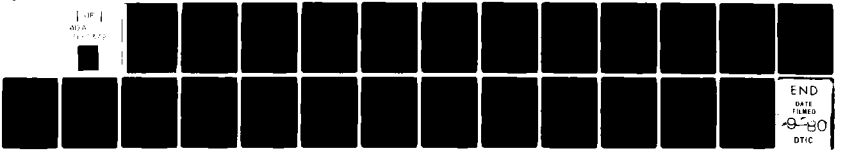
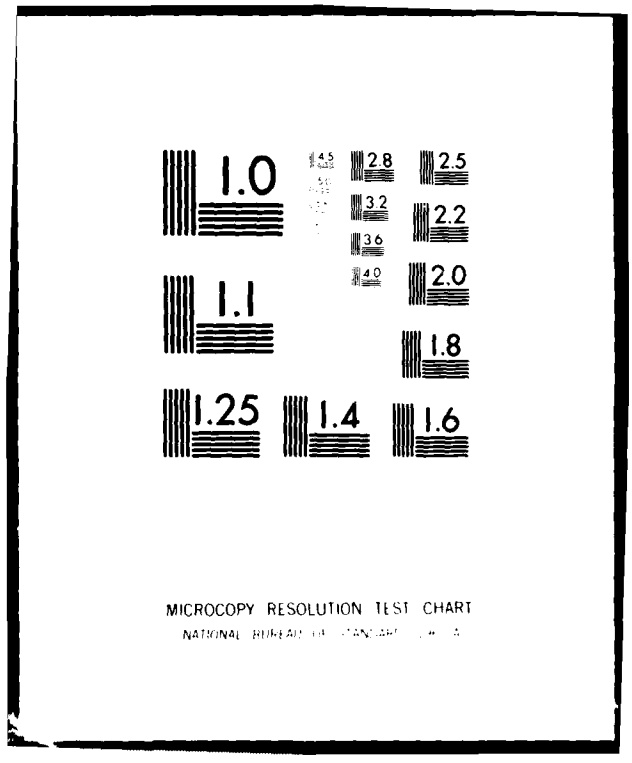
AD-A087 372 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
LOOP ITERATION MACRO,(U)
JUL 80 G BURKE, D MOON N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TM-169 NL



END
DATE FILMED
9-80
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-169

LOOP ITERATION MACRO

Glenn Burke
David Moon

July 1980

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Support for this research was provided in part by National Institutes of Health grant number 1 P41 RR 01096-04 from the Division of Research Resources, and the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract number N00014-75-C-0661.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/TM-169	2. GOVT ACCESSION NO.: AD-A087372
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): LOOP Iteration Macro	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TM-169
7. AUTHOR(s): Glenn Burke and David Moon	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0661, 1-P41-RR-01096-04
9. PERFORMING ORGANIZATION NAME AND ADDRESS: MIT/Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: PHS
11. CONTROLLING OFFICE NAME AND ADDRESS: ONR/Department of the Navy, Information Systems Program, Arlington, VA 22217	12. REPORT DATE: July 1980 13. NUMBER OF PAGES: 24
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Iteration
Lisp
Macro

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

LOOP is a Lisp macro which provides a programmable iteration facility. The same LOOP module operates compatibly in both Lisp Machine Lisp and Maclisp (PDP-10 and Multics). LOOP was inspired by the "FOR" facility of CLISP in InterLisp; however, it is not compatible and differs in several details.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

409648

LOOP Iteration Macro

July 1980

Glenn Burke
David Moon

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Support for this research was provided in part by National Institutes of Health grant number 1 P41 RR 01096-04 from the Division of Research Resources, and the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract number N00014-75-C-0661.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE **MASSACHUSETTS 02139**

Abstract

LOOP is a Lisp macro which provides a programmable iteration facility. The same LOOP module operates compatibly in both Lisp Machine Lisp and Maclisp (PDP-10 and Multics). LOOP was inspired by the "FOR" facility of CLISP in InterLisp; however, it is not compatible and differs in several details.

Any comments, suggestions, or criticisms will be welcomed. The authors can be reached by any of the following communication paths:

ARPA Network mail to BUG-LOOP@MIT-ML

U.S. Mail to
Glenn S. Burke or David A. Moon
Laboratory for Computer Science
545 Technology Square
Cambridge, Mass. 02139

There is also an ARPA Network mail distribution list for announcements pertaining to LOOP. Contact the authors as above to be placed on it.

Acknowledgements

Thanks goes to Peter Szolovits, who implemented FOR, the predecessor of LOOP, and to Lowell Hawkinson, for helping in the design process. The people of the Clinical Decision Making, Lisp Machine, and Knowledge Based Systems groups all deserve note for their use of LOOP early in its development, thus aiding both design and debugging; especially helpful were John Kulp, William Long, William Martin, and Ramesh Patil.

Key Words: Iteration, Lisp, Macro

Table of Contents

1. Introduction

LOOP is a Lisp macro which provides a programmable iteration facility. The same LOOP module operates compatibly in both Lisp Machine Lisp and Maclisp (PDP-10 and Multics). LOOP was inspired by the "FOR" facility of CLISP in InterLisp; however, it is not compatible and differs in several details.

The general approach is that a form introduced by the word **loop** generates a single program loop, into which a large variety of features can be incorporated. The loop consists of some initialization (*prologue*) code, a body which may be executed several times, and some exit (*epilogue*) code. Variables may be declared local to the loop. The features are concerned with loop variables, deciding when to end the iteration, putting user-written code into the loop, returning a value from the construct, and iterating a variable through various real or virtual sets of values.

The **loop** form consists of a series of clauses, each introduced by a keyword symbol. Forms appearing in or implied by the clauses of a **loop** form are classed as those to be executed as initialization code, body code, and/or exit code, but aside from that they are executed strictly in the order implied by the original composition. Thus, just as in ordinary Lisp code, side-effects may be used, and one piece of code may depend on following another for its proper operation. This is the principal philosophy difference from InterLisp's "FOR" facility.

Note that **loop** forms are intended to look like stylized English rather than Lisp code. There is a notably low density of parentheses, and many of the keywords are accepted in several synonymous forms to allow writing of more euphonious and grammatical English. Some find this notation verbose and distasteful, while others find it flexible and convenient. The former are invited to stick to **do**.

```
(defun print-elements-of-list (list-of-elements)
    (loop for element in list-of-elements
          do (print element)))
```

The above function prints each element in its argument, which should be a list. It returns **nil**.

```
(defun extract-interesting-numbers (start-value end-value)
   (loop for number from start-value to end-value
         when (interesting-p number) collect number))
```

The above function takes two arguments, which should be fixnums, and returns a list of all the numbers in that range (inclusive) which satisfy the predicate **interesting-p**.

```
(defun find-maximum-element (array)
   (loop for i from 0 below (cadr (arraydims array))
         maximize (funcall array i)))
```

Find-maximum-element returns the maximum of the elements of its argument, a one-dimensional array.

```
(defun remove (object list)
   (loop for element in list
         unless (equal object element) collect element))
```

Remove is like the Lisp function delete, except that it copies the list rather than destructively splicing out elements.

```
(defun find-frob (list)
   (loop for element in list
         when (frobp element) return element
         finally (error '|Frob not found in list| list)))
```

This returns the first element of its list argument which satisfies the predicate frobp. If none is found, an error is generated.

2. Clauses

Internally, LOOP constructs a prog which includes variable bindings, pre-iteration (initialization) code, post-iteration (exit) code, the body of the iteration, and stepping of variables of iteration to their next values (which happens on every iteration after executing the body).

A *clause* consists of the keyword symbol and any other Lisp forms and keywords which it deals with. For example,

`(loop for x in l do (print x))`,

contains two clauses, "for x in l" and "do (print x)". Certain of the parts of the clause will be described as being *expressions*, e.g. "(print x)" in the above. An expression can be a single Lisp form, or a series of forms implicitly collected with progn. An expression is terminated by the next following atom, which is taken to be a keyword. Thus, syntax allows only the first form in an expression to be atomic, but makes misspelled keywords more easily detectable.

Bindings and iteration variable steppings may be performed either sequentially or in parallel, which affects how the stepping of one iteration variable may depend on the value of another. The syntax for distinguishing the two will be described with the corresponding clauses. When a set of things is "in parallel", all of the bindings produced will be performed in parallel by a single lambda binding. Subsequent bindings will be performed inside of that binding environment.

2.1 Iteration-Producing Clauses

These clauses all create a *variable of iteration*, which is bound locally to the loop and takes on a new value on each successive iteration. Note that if more than one iteration-producing clause is used in the same loop, several variables are created which all step together through their values; when any of the iterations terminates, the entire loop terminates. Nested iterations are not generated; for those, you need a second loop form in the body of the loop.

All of the iteration-producing clauses initially defined are introduced with the keyword for (or as, which is synonomous). For clauses may be clustered into groups, the variables of iteration of which are to be stepped in parallel, by introducing the additional clauses with and instead of for or as. For example, the following iterates over the elements in a list, and also has a variable for the element from the previous iteration:

```
(loop for item in list and previous-item = 'foo then item
      do ...)
```

During the first iteration, previous-item has the value foo; in subsequent iterations, it has the value of item from the previous iteration. Note that this would not work if the stepping were not performed in parallel.

The order of evaluation in iteration-producing clauses is that those expressions which are only evaluated once are evaluated in order at the beginning of the form, during the variable-binding phase, while those expressions which are evaluated each time around the loop are evaluated in order in the body.

These are the iteration-producing clauses. Optional parts are enclosed in curly brackets.

for *var* {*data-type*} **in** *expr1* {**by** *expr2*}
: This iterates over each of the elements in the list *expr1*. If the by subclause is present, *expr2* is evaluated once on entry to the loop to supply the function to be used to fetch successive sublists, instead of cdr.

for *var* **on** *expr1* {**by** *expr2*}
: This is like the previous for format, except that *var* is set to successive tails of the list instead of successive elements.

for *var* {*data-type*} **=** *expr*
: On each iteration, *expr* is evaluated and *var* is set to the result.

for *var* {*data-type*} **=** *expr1* **then** *expr2*
: *Var* is bound to *expr1* when the loop is entered, and set to *expr2* on all succeeding iterations.

for *var* {*data-type*} **from** *expr1* {**to** *expr2*} {**by** *expr3*}
: This performs numeric iteration. *Var* is initialized to *expr1*, and on each succeeding iteration is incremented by *expr3* (default 1). If the to phrase is given, the iteration terminates when *var* becomes greater than *expr2*. Each of the expressions is evaluated only once, and the to and by phrases may be written in either order. Downto may be used instead of to, in which case *var* is decremented by the step value, and the endtest is adjusted accordingly. If below is used instead of to, or above instead of downto, the iteration will be terminated

before *expr2* is reached, rather than after. Note that the to variant appropriate for the direction of stepping must be used for the endtest to be formed correctly, i.e. the code will not work if *expr3* is negative or zero. If no limit specifying clause is given, then the direction of the stepping may be specified as being decreasing by using downfrom instead of from. Upfrom may also be used instead of from; it forces the stepping direction to be increasing. The *data-type* defaults to fixnum.

for *var* {*data-type*} being *expr* and its *path* ...
for *var* {*data-type*} being {each} *path* ...

This provides a user-definable iteration facility. *Path* names the manner in which the iteration is to be performed. The ellipsis indicates where various path dependent preposition/expression pairs may appear. See the section on Iteration Paths (page 11) for complete documentation.

2.2 Bindings

The with keyword may be used to establish initial bindings, that is, variables which are local to the loop but are only set once, rather than on each iteration. The with clause looks like:

with *var1* {*data-type*} {= *expr1*}
{and *var2* {*data-type*} {= *expr2*}}...

If no *expr* is given, the variable is initialized to the appropriate value for its data type, usually nil.

With bindings linked by and are performed in parallel; those not linked are performed sequentially. That is,

```
(loop with a = (foo) and b = (bar) ...)
```

binds the variables like

```
((lambda (a b) ...)
 (foo) (bar))
```

whereas

```
(loop with a = (foo) with b = (barprime a) ...)
```

binds the variables like

```
((lambda (a)
    ((lambda (b) ...)
     (barprime a)))
 (foo))
```

All *expr*'s in with clauses are evaluated in the order they are written, upon entrance to the loop rather than where they appear in the body. Thus good style suggests that with clauses be placed first in the loop.

For binding more than one variable with no particular initialization, one may use the construct

with *variable-list* {*data-type-list*} {and ...}

as in

```
with (i j k t1 t2) (fixnum fixnum fixnum) ...
```
which is a useful special case of *destructuring* (page 10).

2.3 Entrance and Exit

initially *expression*

This puts *expression* into the *prologue* of the iteration. It will be evaluated before any other initialization code other than the initial bindings. For the sake of good style, the **initially** clause should therefore be placed after any **with** clauses but before the main body of the loop.

finally *expression*

This puts *expression* into the *epilogue* of the loop, which is evaluated when the iteration terminates (other than by an explicit **return**). For stylistic reasons, then, this clause should appear last in the loop body. Note that certain clauses may generate code which terminates the iteration without running the epilogue code; this behaviour is noted with those clauses.

2.4 Side Effects

do *expression*
doing *expression*

Expression is evaluated each time through the loop.

2.5 Values

The following clauses accumulate a return value for the iteration in some manner. The general form is

type-of-collection expr {*data-type*} {**into** *var*}

where *type-of-collection* is a **loop** keyword, and *expr* is the thing being "accumulated" somehow. If no **into** is specified, then the accumulation will be returned when the **loop** terminates. If there is an **into**, then when the epilogue of the **loop** is reached, *var* (a variable automatically bound locally in the loop) will have been set to the accumulated result and may be used by the epilogue code. In this way, a user may accumulate and somehow pass back multiple values from a single **loop**, or use them during the loop. It is safe to reference these variables during the loop, but they should not be modified until the epilogue code of the loop is reached. For example,

```
(loop for x in list
      collect (foo x) into foo-list
      collect (bar x) into bar-list
      collect (baz x) into baz-list
     finally (return (list foo-list bar-list baz-list)))
```
which has the same effect as

```
(do ((g0001 l (cdr g0001)) (x) (foo-list) (bar-list) (baz-list))
    ((null g0001)
     (list (nreverse foo-list)
           (nreverse bar-list)
           (nreverse baz-list)))
  (setq x (car g0001))
  (setq foo-list (cons (foo x) foo-list))
  (setq bar-list (cons (bar x) bar-list))
  (setq baz-list (cons (baz x) baz-list)))
```

collect *expr* {**into** *var*}
collecting ...
This causes the values of *expr* on each iteration to be collected into a list.

nconc *expr* {**into** *var*}
nconcing ...
append ...
appending ...
These are like **collect**, but the results are **nconc**ed or **append**ed together as appropriate. **collecting** : **mapcar** :: **nconcing** : **mapcan**.

count *expr* {**into** *var*}
counting ...
If *expr* evaluates non-**nil**, a counter is incremented. The *data-type* is always **fixnum**.

sum *expr* {*data-type*} {**into** *var*}
summing ...
Evaluates *expr* on each iteration, and accumulates the sum of all the values. *Data-type* defaults to **number**, which for all practical purposes is **notype**.

maximize *expr* {*data-type*} {**into** *var*}
minimize ...
Computes the maximum (or minimum) of *expr* over all iterations. *Data-type* defaults to **number**.

Not only may there be multiple *accumulations* in a **loop**, but a single *accumulation* may come from multiple places *within the same* **loop** *form*. Obviously, the types of the collection must be compatible. **Collect**, **nconc**, and **append** may all be mixed, as may **sum** and **count**, and **maximize** and **minimize**. For example,

```
(loop for x in '(a b c) for y in '((1 2) (3 4) (5 6))
      collect x
      append y)
=> (a 1 2 b 3 4 c 5 6)
```

The following computes the average of the entries in the list *list-of-frobs*:

```
(loop for x in list-of-frobs
      count t into count-var
      sum x into sum-var
     finally (return (quotient sum-var count-var)))
```

2.6 Endtests

The following clauses may be used to provide additional control over when the iteration gets terminated, possibly causing exit code (due to finally) to be performed and possibly returning a value (e.g., from collect).

while *expr*
: If *expr* evaluates to nil, the loop is exited, performing exit code (if any), and returning any accumulated value. The test is placed in the body of the loop where it is written. It may appear between sequential for clauses.

until *expr*
: Identical to while (not *expr*).

This may be needed, for example, to step through a strange data structure, as in

```
(loop for concept = expr then (superior-concept concept)
      until (eq concept [summum-genus])
      ...)
```

2.7 Aggregated Boolean Tests

always *expr*
: If *expr* evaluates to nil, the iteration is terminated and nil returned; otherwise, t will be returned when the loop finishes, after the epilogue code (if any, as specified with the finally clause) has been run.

never *expr*
: This is like always (not *expr*).

thereis *expr*
: If *expr* evaluates non-nil, then the iteration is terminated and that value is returned, without running the epilogue code.

2.8 Conditionalization

These clauses may be used to "conditionalize" the following clause. They may precede any of the side-effecting or value-producing clauses, such as **do**, **collect**, or **always**.

when *expr*
if *expr*
: If *expr* evaluates to nil, the following clause will be skipped, otherwise not.

unless *expr*
: This is equivalent to **when (not** *expr*).

Multiple conditionalization clauses may appear in sequence. If one test fails, then any following tests in the immediate sequence, and the clause being conditionalized, are skipped.

Multiple clauses may be conditionalized under the same test by joining them with **and**, as in

```
(loop for i from a to b
      when (zerop (remainder i 3))
      collect i and do (print i))
```

which returns a list of all multiples of 3 from *a* to *b* (inclusive) and prints them as they are being collected.

Conditionals may be nested. For example,

```
(loop for i from a to b
      when (zerop (remainder i 3))
      do (print i)
      and when (zerop (remainder i 2))
          collect i)
```

returns a list of all multiples of 6 from *a* to *b*, and prints all multiples of 3 from *a* to *b*.

Useful with the conditionalization clauses is the **return** clause, which causes an explicit return of its "argument" as the value of the iteration, bypassing any epilogue code. That is,

when *expr1* **return** *expr2*

is equivalent to

when *expr1* **do (return** *expr2*)

Conditionalization of one of the "aggregated boolean value" clauses simply causes the test which would cause the iteration to terminate early not to be performed unless the condition succeeds. For example,

```
(loop for x in l
      when (significant-p x)
        do (print x) (princ "is significant.")
        and thereis (extra-special-significant-p x))
```

The format of a conditionalization and following clause is typically something like

when *expr1 keyword expr2*

If *expr2* is the keyword **it**, then a variable is generated to hold the value of *expr1*, and that variable gets substituted for *expr2*. Thus, the composition

when *expr* **return it**

is equivalent to the clause

thereis *expr*

and one may collect all non-null values in an iteration by saying

when *expression* **collect it**

If multiple clauses are joined with **and**, the **it** keyword may only be used in the first. If multiple whens, unlesses, and/or ifs occur in sequence, the value substituted for **it** will be that of the last test performed.

3. LOOP Synonyms

define-loop-macro *Macro*

```
(define-loop-macro keyword)
```

may be used to make *keyword*, a **loop** keyword (such as **for**), into a LISP macro which may introduce a **loop** form. For example, after evaluating

```
(define-loop-macro for),
```

one may now write an iteration as

```
(for i from 1 below n do ...)
```

4. Data Types

In many of the clause descriptions, an optional *data-type* is shown. A *data-type* in this sense is an atomic symbol, and is recognizable as such by LOOP. LOOP interfaces to a module which defines how declarations and initializations are to be performed for various data types. However, it recognizes several types specially so that that module need not be present in order for them to be used:

fixnum
: An implementation-dependent limited range integer.

flonum
: An implementation-dependent limited precision floating point number.

integer
: Any integer (no range restriction).

number
: Any number.

notype
: Unspecified type (i.e., anything else).

5. Destructuring

Destructuring provides one with the ability to "simultaneously" assign or bind multiple variables to components of some data structure. Typically this is used with list structure (which is the only mode currently supported). For example,

```
(desetq (foo . bar) '(a b c))
```

has the effect of setting foo to a and bar to (b c). LOOP only requires destructuring support when one of these patterns is supplied in place of a variable. In addition, the "binding" of a pattern to a constant nil is so treated that it requires no special support code; this allows the case

```
with (a b c)
```

to work without destructuring support code.

One may specify the data types of the components of a pattern by using a corresponding pattern of the data type keywords in place of a single data type keyword. This syntax remains unambiguous because wherever a data type keyword is possible, a loop keyword is the only other possibility. Thus, if one wants to do

```
(loop for x in l
      as i fixnum = (car x)
         and j fixnum = (cadr x)
         and k fixnum = (cddr x)
      ...)
```

and no reference to x is needed, one may instead write

```
(loop for (i j . k) (fixnum fixnum . fixnum) in l ...)
```

To allow some abbreviation of the data type pattern, an atomic data type component of the pattern is considered to state that all components of the corresponding part of the variable pattern are of that type. That is, the previous form could be written as

```
(loop for (i j . k) fixnum in l ...)
```

This generality allows binding of multiple typed variables in a reasonably concise manner, as in

```
(loop with (a b c) and (i j k) fixnum ...)
```

which binds a, b, and c to nil and i, j, and k to 0 for use as temporaries during the iteration, and declares i, j, and k to be fixnums for the benefit of the compiler.

```
(defun map-over-properties (fn symbol)
  (loop for (propname propval) on (plist symbol) by 'cddr
        do (funcall fn symbol propname propval)))
```

See also section 8, page 17, which discusses support code needed in various implementations.

6. Iteration Paths

Iteration paths provide a mechanism for user extension of iteration-producing clauses. The interface is constrained so that the definition of a path need not depend on much of the internals of LOOP. In general, a path iteration has one of the forms

```
for var {data-type} being expr0 and its pathname
    {preposition1 expr1}...
for var {data-type} being {each} pathname of expr0
    {preposition1 expr1}
```

The difference between the two is this: in the first, *var* will take on the value of *expr0* the first time through the loop; but in the second, it will be the "first step along the path". *Pathname* is an atomic symbol which is defined as a loop path function. The usage and defaulting of *data-type* is up to the path function. Any number of preposition/expression pairs may be present; the prepositions allowable for any particular path are defined by that path. The of preposition has special meaning in that it specifies the starting point of the path; thus, the first variation shown implicitly uses an of *expr0* "prepositional phrase". To enhance readability, pathnames are usually defined in both the singular and plural forms. To satisfy the anthropomorphic among you, his, her, or their may be substituted for the its keyword. Egocentricity is not condoned.

One pre-defined path is cars; it simply iterates over successive cars of its starting argument, terminating after an atom is reached. For example,

```
(loop for x being cars of '((a b) c) collect x)
  => ((a b) a)
(loop for x being '((a b) c) and its cars collect x)
  => (((a b) c) (a b) a)
```

The above forms are equivalent to

```
(loop for x = (car '((a b) c)) then (car x)
      collect x
      until (atom x))
```

and

```
(loop for x = '((a b) c) then (car x)
      collect x
      until (atom x))
```

respectively. (Note that the atom check following the body of this loop is part of the definition of the cars path, and is not a property of paths in general.)

By special dispensation, if a *pathname* is not recognized, then the attachments path will be invoked upon a syntactic transformation of the original input. This name derives historically from its original usage in XLMS. Essentially, the loop fragment

```
for var being e-r of expr ...
```

is taken as if it were

```
for var being attachments in e-r-* of expr ...
```

and

```
for var being expr and its e-r ...
```

is taken as if it were

```
for var being expr and its attachments in e-r-*
```

Thus, this "undefined pathname hook" only works if the attachments path is defined. Note also:

loop-attachment-transformer *Variable*

The value of this is a function of one argument which will be called on *e-r* to transform it into *e-r-**. If it is nil, then a quote is listed around the expression, effectively causing the special attachments syntax to be an unevaluated form of the attachments path. This is initially nil except in an LMS environment, in which case it is a function which simply returns *e-r*.

6.1 Defining Paths

This section will probably be of interest only to those interested in defining their own paths.

For the purposes of discussion, the general template form of an iteration may be assumed to be

```
(let variable-bindings
  (prog ()
    prologue-code
   next-loop
    pre-body-endtests-1
    pre-body-steps-1
    pre-body-endtests-2
    pre-body-steps-2
    ...
    body
    post-body-endtests-1
    post-body-steps-1
    post-body-endtests-2
    post-body-steps-2
    ...
    (go next-loop)
   end-loop
    epilogue-code
    ))
```

When more than one for clause is grouped together with and, the endtests and steps are arranged to occur together in parallel. Sequentially arranged for clauses cause multiple endtests and steps to occur one after another, as shown in the above template.

A function to generate code for a path may be declared to loop with the define-loop-path function:

define-loop-path *pathname-or-names path-function list-of-allowable-prepositions (any-number-of data)*

This defines *path-function* to be the handler for the path(s) *pathname-or-names*, which may be either a symbol or a list of symbols. Such a handler should follow the conventions described below.

The handler will be called with the following arguments:

path-name

The name of the path which caused the path function to be invoked.

variable

The "iteration variable".

data-type

The data type supplied with the iteration variable, or nil if none was supplied.

prepositional-phrases
This is a list with entries of the form *(preposition expression)*, in the order in which they were collected. This may also include some supplied implicitly (e.g. of phrases, and in phrases for attachment relations); the ordering will show the order of evaluation which should be followed for the expressions.

inclusive?
This is t if *variable* should have the starting point of the path as its value on the first iteration, nil otherwise.

allowed-prepositions
This is the list of allowable prepositions declared for the pathname that caused the path function to be invoked. It and *data* (immediately below) may be used by the path function such that a single function may handle similar paths.

data This is the list of "data" declared for the pathname that caused the path function to be invoked. It may, for instance, contain a canonicalized pathname, or a set of functions or flags to aid the path function in determining what to do. In this way, the same path function may be able to handle different paths.

The handler should return a list with the following elements:

variable-bindings
This is a list of variables which need to be bound. The entries in it may be of the form *variable*, (*variable expression*), or (*variable expression data-type*). Note that it is the responsibility of the handler to make sure the iteration variable gets bound. All of these variables will be bound in parallel; thus, if initialization of one depends on others, it should be done with a setq in the *prologue-forms*.

prologue-forms
This is a list of forms which should be included in the loop prologue.

pre-body-endtest
This is a single form.

pre-body-steps
This should be an alternating list of variables and expressions to step them. They will be stepped in parallel. (This is like the arguments to setq; in fact, it will be used as the arguments to psetq.)

post-body-endtest
Like *pre-body-endtest*, but done after the *body*, just before starting the next iteration.

post-body-steps
Like *pre-body-steps*.

If anyone finds that they need to modify the *main* body or the epilogue code, we would like to hear about it.

A qualification is in order with respect to stepping. In order to make parallel stepping work properly, loop must be able to coerce the stepping code for different for clauses to act in parallel. Thus, the canonical place for stepping to occur is in the *post-body-steps*; the

pre-body-steps is mainly useful when the iteration variable needs to be set to some function of whatever is actually being iterated over. For example, the LOOP clause

for *var* in *list*

effectively returns the following elements for the template (where *tem* is really a gensymed variable name):

variable-bindings
(*var* (*tem list*))

prologue-forms
nil

pre-body-endtest
(null *tem*)

pre-body-steps
(*var* (car *tem*))

post-body-endtest
nil

post-body-steps
(*tem* (cdr *tem*))

loop-tequal *token symbol-or-string*

This is the LOOP token comparison function. *Token* is any Lisp object; *symbol-or-string* is the keyword it is to be compared against. It returns t if they represent the same token, comparing in a manner appropriate for the implementation. In certain implementations **loop-tequal** may be a macro.

7. Compatibility with FOR

LOOP is not truly compatible with FOR (a similar Maclisp iteration package). The reason for this is that LOOP has certain "ideas" about how it should handle such things as order of evaluation and repeated evaluation, which are quite different from FOR's simpler template approach. Many of the keywords, and hopefully all of the functionality, have been preserved. In many cases, code written with FOR will work with LOOP, although it sometimes may not behave identically. For convenience, here is a (non-exhaustive) summary of the major differences.

One major difference is that LOOP is more fastidious about how it orders the assignments and endtests. Take, for example

```
(loop for n in list as z = (* n n) collect z)
```

In FOR, n would be assigned to the car of the list, then z would be stepped, and then the null check would be made on the iteration list. This means that on the last iteration z will be assigned to (* nil nil), which might cause some consternation to the Lisp interpreter. In LOOP, first a null check is made on the list, then n is set to the car of the list, then z is stepped.

Explicit endtests (while and until) are placed "where they appear" in the iteration sequence. This obviates the repeat-while and repeat-until keywords of FOR. For example, the FOR construct

```
(for x in l collect x repeat-while (< x 259.))
```

may be replaced by the LOOP construct

```
(loop for x in l collect x while (< x 259.))
```

Note that in the FOR case, the ordering of the clauses typically does not matter, but in the LOOP case it typically does. Thus, the ordering in

```
(loop for data = (generate-some-data)
      collect (f data)
      while (test data))
```

causes the result to be a list with at least one element.

LOOP attempts to suppress repeated evaluation where possible. Which expressions get repeatedly evaluated is documented with the corresponding clauses. One significant example where LOOP and FOR differ is in the case

```
(loop for i from 0 to expression ...)
```

in which FOR evaluates *expression* at every iteration, whereas LOOP saves the value at the start of the iteration.

It should be noted that the conditionalization clauses (when, until, and if) affect only the following clause rather than the whole of the "body" of the iteration, as would be the case in FOR.

Because it is difficult for it to work in all cases, the trailing clause has been eliminated. Its effect may be achieved, however, by tacking

```
and var = initial-value then var-to-be-trailed
```

after the for clause which steps *var-to-be-trailed*.

8. Dependencies

The LOOP package may require the existence of other routines in some implementations. For efficiency reasons, LOOP avoids producing **let** in the code it generates unless it is necessary for destructuring bindings.

In the PDP-10 Maclisp implementation, LOOP uses **ferror** to generate error messages; **ferror** is part of the FORMAT package, and is assumed to be autoloadable from there. **Let**, which is used to produce destructuring bindings, and the destructuring version of **setq** called **desetq**, which is used only when destructuring is used, are both autoloadable. The "parallel setq" mechanism is simulated so that **psetq** is not needed. Macro memoizing is performed using the same facilities which **defmacro** uses, and are autoloadable (and typically present in most environments).

In Multics Maclisp, LOOP does not presently call **ferror**, which does not exist. There is a **let** macro available with destructuring capability; it is non-standard (not part of the Multics Lisp system) — for further information contact the authors. Currently, macro memoizing is performed by **rplaca/rplacd** splicing, unconditionally.

In Lisp Machine lisp, **ferror** is used to generate errors. This is part of the basic Lisp Machine environment. At this time, destructuring support is not part of the basic environment, although it is available; contact either the authors or the Lisp Machine group if you need this. Macro memoizing is performed using **displace**, with the same effect as in Multics Maclisp.

Index

OFFICIAL DISTRIBUTION LIST

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Branch Office/Boston
Building 114, Section D
666 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office/Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office/Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

New York Area
715 Broadway - 5th floor
New York, N. Y. 10003
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375
6 copies

Assistant Chief for Technology
Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Naval Ocean Systems Center, Code 91
Headquarters-Computer Sciences &
Simulation Department
San Diego, CA 92152
Mr. Lloyd Z. Maudlin
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Math Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper, USNR
NAVDAC-OOH
Department of the Navy
Washingon, D. C. 20374
1 copy